An Introduction To Attribute Grammars

Sven Karol
Department of Computer Science
Technische Universität Dresden
Germany
Sven.Karol@mailbox.tu-dresden.de

ABSTRACT

Beside the syntax, semantic is a very important part of programming languages. Without a semantic a program would no longer be a program but a lifeless sequence of characters which is part of the language.
The dynamic semantic properties of a program are determined during execution at runtime. Typically they directly depend on the input values of the program. By contrast the static semantic properties of a program are computed at compile-time when it is translated from source code representation into some kind of goal representation. The static semantic of a programming language is used to determine those properties.

The concept of attributed context-free grammars (Attribute Grammars) addresses this aspect. It enables a user to create complete specifications of static semantics. This article gives a short introduction on the topic of attributed context-free grammars. It shows how to define such a construct and introduces the concept of synthesised and inherited attributes which can be associated to grammar symbols. Furthermore it will be shown how semantic rules on attributes can be defined. The final part of the paper gives an introduction into different types of attribute grammars and attribute evaluators.

1. INTRODUCTION

Attribute Grammars(AGs) are a widely known approach to express the static semantics of programming languages. They were first introduced as a formal mechanism by Donald E. Knuth in 1968 [1]. Previously they already have been used informally by many compiler developers to define the static semantics of a programming language: In the 1960s there has been a big discussion about how to specify the semantics of context-free languages. Many experts tried out to find a declarative concept but did not succeed while computers went better, and programming languages went more complex. Hence the compiler parts which dealt with the semantics began to become very complex and nearly unmaintainable.
Finally, in 1967 [4], Knuth realised that many people used the same concept of attributes in compilers and that there are especially attributes which only flow from top-down or bottom-up through a parse tree - the idea idea of attribute grammars was born.
Today most compiler-generators use AGs to generate the components for the semantics analysis phase out of a user's specification automatically. In the process of compilation the semantics analysis is the third phase following lexical and syntactical analysis which only deal with context free (syntactic) properties of a language. A lexical analyser(lexer) converts an input stream of characters into a stream of tokens or rather, a stream of terminal symbols. Tokens are the smallest unit a parser can handle. Hence a syntactical analyser(parser) converts an input stream of tokens into a(n) (attributed) syntax tree. The third phase addresses context dependent properties, especially those which carry static semantic. So, in the case of attribute grammars, a semantic analyser(attribute evaluator) takes an unevaluated attributed syntax tree as input and has the evaluated attributed syntax tree as output. Note that typically an attribute evaluator is not the only part of the semantic analysis phase.

A context dependent property of a programming language has static semantics if it can be computed at compile-time of a program. Such a property might be the type information of variables or the result type of arithmetic expressions. In difference to that, a property with dynamic semantic must be computable during execution at runtime. Hence such properties often depend on the input values of a program directly and might change during multiple program executions.

The first of the next three sections deals with the notation and the definition of Attribute Grammars and shows how a context free grammar can be enriched by semantic constructs. Afterwards a simple example is introduced which is used in the whole article to enhance the readers understanding of the different topics. The second section deals with the circularity of Attribute Grammars. Different ways for detecting cycles in *AG*s are explained. The standard approaches for attribute evaluators presented in the third section have no abilities to evaluate *AG*s containing cycles. Hence it might be useful to find them before an evaluation takes place. The last section gives an overview on dynamic and static attribute evaluators. Additionally it introduces L-

attributed grammars as a more restrictive class of Attribute Grammars.
I would recommend readers to be at least familiar with the concept of context free grammars and top-down LL parsing approaches.

2. DEFINING ATTRIBUTE GRAMMARS

The concept of AGs extends the notion of context free grammars through two different kinds of attributes. Inherited attributes are used to specify the flow of information from a node in the abstract syntax tree top-down to lower nodes. Contrary synthesised attributes characterise an information flow in bottom-up direction. Relations between attribute values of different nodes are expressed with semantic rules. Let $G = (N, T, P, S)$ be a context-free grammar with

N - a set of non-terminal symbols,
T - a set of terminal symbols,
P - a set of grammar production rules,
S - the starting symbol,
and $N \cap T = \emptyset$, $p \in P$: $X_0 \rightarrow X_1 \dots X_i \dots X_{np}$ ($X_0 \in N, X_i \in (N \cup T)$), $S \in N$.

The value np represents the count of production elements on a grammar-rule's right side. It might be equal to 0 so that the right side is empty. Such a production is called an ϵ-production deducing the empty word. In the following we use $V = N \cup T$ to represent the grammar vocabulary.

For defining an Attribute Grammar we have to extend the context-free notation with

INH - a set of inherited attributes,
SYN - a set of synthesised attributes,
INH_X - a set of inherited attributes for $X \in V$,
SYN_X - a set of synthesised attributes for $X \in V$,
f - semantic rules bound to syntactic rules and attributes,
and $INH \cap SYN = \emptyset$, $INH_X \subseteq INH$, $SYN_X \subseteq SYN$. Additionally every attribute $a \in INH \cup SYN$ has to be associated with a range of values T_a which one could imagine as type T of a.

Defining f formally is slightly more difficult. If $p = (X_0 \rightarrow X_1 \dots X_i \dots X_{np})$ is a production of the context-free grammar then f has to be defined for every $a_0 \in SYN_{X_0}$ and it has to be defined for every $a_i \in INH_{X_i} (1 \leq i \leq np)$. The arguments of f might consist of any value of any attribute of any grammar symbol in p. The notation $f_{(p,i,a)}$ denotes that a definition of f evaluates the attribute a of the grammar symbol on the ith position in the production p. What does that mean? If $p2$ is a production like $A \rightarrow XA$ in any AG G with

$$INH_A = \{i1\}$$
$$INH_X = \{i2\}$$
$$SYN_A = \{s1\}$$
$$SYN_X = \{s2\}$$

then exactly the semantic rules $f_{(p2,0,s1)}$, $f_{(p2,1,i2)}$ and $f_{(p2,2,i1)}$ have to be defined for $p2$ and none else.

The evaluation of attributes takes place on the abstract syntax tree (AST) which can be regarded as a composition of elemental trees. For every production p in G an elemental tree t_p can be created whose root is X_0 and whose leaves are $\{X_i | 1 \leq i \leq np\}$ (fig. 1).
So any possible AST can be created by repeatedly merging leaves of a subtree with the root of an elemental tree which is labelled by the same non-terminal symbol. In most compilers the parser does this work.

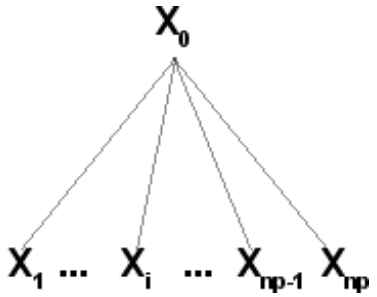

Figure 1: an elemental tree

Every production p of a grammar occurs only once but t_p can occur multiple times in an AST. Hence all occurrences embody the same attributes but different attribute exemplars. This means that similar nodes carry different attribute values.

Now we can use the definition from above to construct a concrete instance of an AG. The example will contain a context free grammar which describes a potential representation of hexadecimal numbers enriched by semantic information for computing the decimal value of those numbers. It is a variation of the example binary notation used by [1]. The form of notation leans against the syntax for AGs used in [2].

attribute grammar AG_{hex}
nonterminals

$$N = \{NUMBER, NUM1, NUM2, HEX\}$$
$$S = NUMBER$$

terminals

$$T = \{., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

attributes

$$INH = \{position : int\}$$
$$SYN = \{dvalue : real\}$$
$$SYN_{NUMBER} = \{dvalue\},\ INH_{NUMBER} = \emptyset$$
$$SYN_{NUM1} = SYN_{NUM2} = \{dvalue\},\ INH_{NUM1} = INH_{NUM2} = \{position\}$$
$$SYN_{HEX} = \{dvalue\},\ INH_{HEX} = \emptyset$$

rules

r1: $NUMBER \rightarrow NUM1$
$NUMBER.dvalue = NUM1.dvalue$

$NUM1.position = 0$

r2: $NUMBER \rightarrow NUM1\ .\ NUM2$
$NUMBER.dvalue = NUM1.dvalue + NUM2.dvalue$
$NUM1.position = 0$
$NUM2.position = 1$

r3: $NUM1 \rightarrow NUM1\ HEX$
$NUM1_0.dvalue = HEX.dvalue * 16^{NUM1_0.position} + NUM1_1.dvalue$
$NUM1_1.position = NUM1_0.position + 1$

r4: $NUM1 \rightarrow \epsilon$
$NUM1.dvalue = 0$

r5: $NUM2 \rightarrow HEX\ NUM2$
$NUM2_0.dvalue = HEX.dvalue * 16^{-NUM2_0.position} + NUM2_1.dvalue$
$NUM2_1.position = NUM2_0.position + 1$

r6: $NUM2 \rightarrow \epsilon$
$NUM2.dvalue = 0$

r7: $HEX \rightarrow \{0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F\}$
$HEX.dvalue = getValueForChar()$

Note that in this example the semantic rules are directly annotated below the syntactic rule they correspond to. In practice semantic rules can become very large thus they normally would be stored in programming-libraries.
The grammar defines a number to consist of one part if it is a whole number or two parts separated by a point if it is a real number. The left-recursive non-terminal $NUM1$ represents the integer part while the right-recursive $NUM2$ is used to represent the real part. It would be possible to use only one of both non-terminals but then one would have to introduce an other inherited attribute counting the length of a side at first.
Hence AG_{hex} uses two attributes only. The synthesised attribute *dvalue* is used to accumulate the computed decimal values of the hexadecimal cyphers. Those values are published by every $NUM1$ or $NUM2$ node in the AST to its predecessor. The attribute *position* is an inherited attribute which is used to count the actual position in a hexadecimal number. It is incremented and published top down to the corresponding $NUM1$ or $NUM2$ node. There it is used in the exponent of the computation of the decimal values. Finally after all attributes have been evaluated the result can be found in the root of the AST.
It is very likely that any meaningful context-free grammar contains productions with multiple occurrences of the same grammar symbol - at least in recursions. So there must be a possibility to distinguish between them. Otherwise it would not be possible to define semantic rules for such productions. Typically this problem is solved by introducing an implicit numeration as it was done in the example. For instance an argument A_0 of a semantic rule would correspond to the first occurence of the grammar-symbol A in a syntactic rule $r = (A \rightarrow XAY)$ while A_1 addresses the second exemplar.
Another interesting property of AG_{hex} can be found. If an attribute occurs on a left side of a grammar production's semantic rules it never occurs on the right side and vice versa. This property is called the **normal form** of an AG. More formally:

Let AG be an attribute grammar and $p \in P$ a production then AG is in **normal form** if all arguments arg of semantic rules can be found in $(INH_{X0} \cup SYN_{Xi})(1 \leq i \leq np)$.

Let us take a look on the concrete evaluated derivation tree T_{ex} for the number string $h =' 08.15'$(fig.2). Obviously 08.15_{hex} seems to be equal to 8.082_{dec} as it is stored in $NUMBER.dvalue$. In this notation synthesised attributes are always noted to the right of a node while inherited attributes are noted to the left.

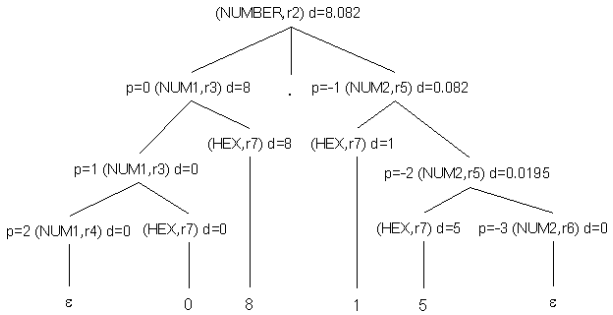

Figure 2: example: attributed abstract syntax tree for h='08.15'

Inherited attributes are not necessarily needed. So for every attribute grammar $A(INH \neq \emptyset)$ it is possible to create at least one equivalent attribute grammar $B(INH = \emptyset)$ by introducing a synthesised attribute c to transfer all information about the nodes in the tree at the root [1]. However a usage of synthesised attributes only often leads to very complicated semantic rules and an additional overhead for the evaluation of the attribute c at the root. Hence typically inherited attributes lead to a better readability and less computation effort.
Nevertheless there are exceptions to that rule. For instance in our example it would be possible to use synthesised attributes without introducing much more effort. Therefore we could invert the recursion in the rules r_3 and r_5. Additionally we would have to convert the position attribute into a synthetic attribute. Furthermore the semantic rules of $r1$ and $r2$, which are responsible for initialization of *position*, have to be replaced by similar methods in r_4 and r_6.

3. DEPENDENCIES AND CYCLES

Testing for circularity is a very important topic for AGs because it is impossible to evaluate attributes which depend on themselves. Fortunately there are methods to pre-calculate dependencies of attribute exemplars at generation time, e.g. when a compiler is generated or written by hand. Hence the information can be used to solve two problems.
First of all a grammar can automatically be checked on errors before anything will be created. So it can be used to support the users of compiler-compilers to solve conflicts in grammars. Secondly a cycle free dependency information may be used to pre-compute an order of evaluation steps for a static attribute evaluator.

For any deducible derivation tree T of a grammar G an

evaluator has to take into account all relations between attribute exemplars. Those relations can be expressed by a *dependency graph* $D(T)$. The nodes of $D(T)$ correspond to attribute exemplars in T and can be written as $N.a$, where N is a node in T while a is an attribute of the grammar symbol being the label of N in T. Let $X1 = label(N1)$ and $X2 = label(N2)$ be the grammar symbols which are label of $N1$ and $N2$, then $D(T)$ contains directed edges from $N1.a1$ to $N2.a2$ if there is a applicable semantic rule $X2.a2 = f(.., X1.a1, ..)$ for a production p with $X1 \in p$ and $X2 \in p$ in this context. According to our example, $D(T_{hex})$ is the graph in figure 3.

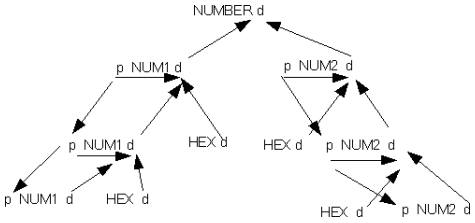

Figure 3: dependency graph $D(T_{hex})$

The transitive closure $D^+(T)$ contains an edge from a node $N1.a1$ to a node $N2.a2$ if, and only if, there is a path between both nodes. If $D^+(T)$ does not contain any edge of the type $(N.a, N.a)$, $D(T)$ contains no cycles and can be evaluated.

An attribute grammar G is **well-formed** if $D(T)$ contains no cycle for every possible derivation tree T.

It is impossible to pre-compute all $D(T)$ because typically it is possible to create an infinite number of derivation trees for G. Hence the dependency information has to be computed directly out of the grammar rules. So it is useful to regard the local dependency graph $D(p)$ for a production p. Let p be a production then $D(p)$ contains nodes $N.a$ for every grammar-symbol $X \in p$ with $X = label(N)$ and all attributes a with $a \in INH_X \cup SYN_X$. There is an arc from $N1.a1$ to $N2.a2$ in $D(p)$ only if there is a semantic rule $X2.a2 = f(.., X1.a1, ..)$ defined for p. $D(p)$ can be constructed for every production in a grammar and any possible $D(T)$ can be created through pasting various $D(p)$ together. Figure 4 shows the $D(p)$s according to our example. None of them contains any cycle. So AG_{hex} is **locally free of cycles** which is the weakest condition for a cycle free grammar: an AG contains no local cycles if every graph in $\{D(p)|p \in P\}$ is free of cycles. As every normalized AG contains no local cycles the test for this property is a co-product of the normalization process [3].

In order to test globally if an attribute grammar contains any cycle some additional constructs are needed [3]. The operation root-projection $rp(D)$ applied on a dependency graph $D(T)$ results in graph which only contains nodes which correspond to the attributes of the underlying tree's root. Additionally $rp(D)$ contains all edges which lead from an attribute of the root node to an attribute of the root node. Another important operation is the overlay operation. Let p be a production, and for $1 \leq i \leq np$ let G_i be any directed graph with nodes $\{a|a \in SYN_{X_i} \cup INH_{X_i}\}$ then

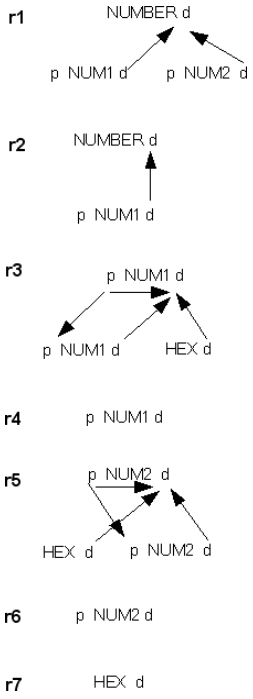

Figure 4: local dependency graphs of AG_{hex}

$D(p)[G_1, \ldots, G_{np}]$ is the local dependency graph for p overlayed with the edges of G_i.
Figure 5 shows both operations applied on some local dependency graphs of our example. The grey edges in that figure do not belong to any graph. They only shall clarify the importance of the order of $D(p)[\ldots]$'s arguments.

With these operations we can calculate sets $S(X)$ for every

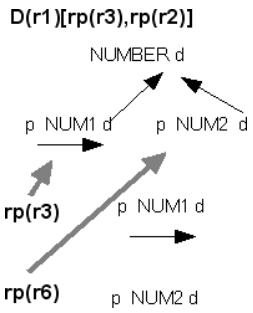

Figure 5: $rp()$ and $D(p)[\ldots]$

grammar-symbol X of an attributed context free grammar. Such a set contains every possible dependency graph between the attributes of X. This information will be enough to determine if any dependency graph $D(T)$ can contain any cycle. We use the following algorithm to compute the $S(X)$:

1. **foreach** $n \in N$: set $S(n) = \emptyset$
2. **foreach** $t \in T$: set $S(t) = \{(SYN_t \cup INH_t, \emptyset)\}$

In other words: Initialize the $S(X)$ for all grammar-symbols. $S(X)$ for non-terminals contains no graph while for terminals $S(X)$ is initialized with a graph having nodes only. Typically terminals are not attributed, hence in most cases $S(X)$

will be empty too.

3. **repeat**

- choose a production $p \in P$
- for $1 \leq i \leq np$ choose $G_i \in S(X_i)$
- compute the overlay $D(p)[G_1, \ldots, G_{np}]$
- compute the root-projection r for the transitive closure $rp(D(p)^+[G_1, \ldots, G_{np}])$
- add r to $S(X_0)$ if, and only if $!(r \in S(X_0))$

until there are no more possibilities to create a new graph which could be added to any $S(X)$.
In other words: Try to construct new graphs by overlaying any rule dependency graph with root-projected graphs which already have been created and add them to $S(X)$. If no new graphs can be added the algorithm terminates.

Now our circularity test is nearly complete. We only have to check if any overlayed graph $D(p)[G_1, \ldots, G_i, \ldots, G_{np}]$ with $G_i \in S(X_i)$ contains any cycle.
If not, the grammar is **free of cycles**. Unfortunately the algorithm seems to have a high time complexity because there are many combination possibilities to create a new graph in $S(X)$. Fortunately there is an other criterion which is even more strict and causes less computation effort. Instead of the set $S(X)$ only one graph $s(X)$ per X can be created. It could be regarded as a merge of all graphs in $S(X)$. For computing the $s(X)$s the above algorithm has to be slightly modified: instead of repeatedly adding graphs to $S(X)$ these will be stepwise merged in $s(X)$ until none of the $s(X)$ can be changed any more. Afterwards we have to check if any overlayed graph $D(p)[G_1, \ldots, G_i, \ldots, G_{np}]$ with $G_i = s(X_i)$ contains any cycle.
If not, the grammar is **absolutely free of cycles**.

It should be said that if an AG is *absolutely free of cycles* the grammar also is *free of cycles* but not vice versa! The test for *absolute cycle freedom* might fail while the standard test for *cycle freedom* does not fail. Hence a test procedure for cycles in a compiler generator might look like in the following:

1. Try to transform the AG into normal form, if that fails return 'grammar contains cycles' otherwise continue with step 2.
2. Apply the test for *absolute cycle freedom*, if that fails continue with step 3 otherwise return 'grammar is free of cycles'.
3. Apply the test for *cycle freedom*, if that fails return 'grammar contains cycles'.

The results of that test do not have to be thrown away. Instead the $s(X)$ could be used by an attribute evaluator at compile-time. Therefore some more theoretical constructs are needed which will not be discussed here. For deepening I would recommend to read [2,chapter 9.4].

4. ATTRIBUTE EVALUATORS

At a glance there are two groups of attribute evaluators. The group of dynamic evaluators computes attribute dependency information on runtime only. In contrast static evaluators use dependency information which has been computed during generation time of a compiler. The basic ideas for that were shown in the last section.
Some of the approaches lead to restrictions on the attribute definition. Hence they also lead to some more restricted classes of AGs.

Data-driven evaluator
This is a very intuitive approach for a dynamical evaluation. It can be applied to all possible attributed context free grammars regardless of if they contain cycles or not - data-driven evaluation can detect cycles dynamically.
An evaluation takes place on the AST where, in every iteration, the algorithm searches for evaluable attributes. Initially those could be found as synthesised attributes of the leafs or on any other node where constant values have been assigned to an attribute. In case of our example this meets attributes of nodes which are evaluated by one of the semantic rules $f_{(r1,1,position)}$, $f_{(r2,1,position)}$, $f_{(r2,2,position)}$, $f_{(r4,0,dvalue)}$, $f_{(r6,0,dvalue)}$ and $f_{(r7,0,dvalue)}$.
The algorithm works as follows:

1. **foreach** node $n \in T$: assign all directly computable values to the corresponding attributes
2. **if** there are more evaluable attributes continue with step 1, **else** goto step 3.
3. **if** there are still unevaluated attributes return 'grammar contains cycle', **else** return 'all attributes evaluated'

Obviously the algorithm has to iterate over all attribute exemplars in the AST during a single pass. In the worst case it finds only one attribute it can evaluate per step. So the time complexity would be c^2 (if c is the number of attributes).

Demand-driven evaluator
This dynamical approach is slightly different from the data-driven approach. It tries to evaluate attributes on the AST regardless if they can be computed directly or depend on other ones. In the case that an attribute depends on others which not have been evaluated too, a demand-driven evaluator tries to evaluate those recursively. Typically the algorithm starts with the synthesised attributes on the root of the AST.
In the abstract syntax tree of our example (fig. 2) the computation could start on the attribute *dvalue* of the $NUMBER$ node.

Due to
$f_{(r1,0,dvalue)} = NUM1.dvalue + NUM2.dvalue$

the algorithm continues with
$f_{(r3,0,dvalue)} = HEX.dvalue * 16^{NUM1_0.position} + NUM1_1.dvalue$

and computes it's first value with
$f_{(r7,0,dvalue)} = getValueForChar()$.

After that the computation continues with the evaluation of $f_{(r3,0,dvalue)}$'s second part. Typically the implementation is a simple recursive function like (for more detail see [3]):

```
public AttrValue Node::eval(Attribute attr){
    if(values.get(attr)!=null)
        return values.get(attr);
    if(working.get(attr)==true)
        throw new Exception("contains cycle");
    working.set(true);
    //find out which semantic rule must be applied
    {....
    //recursion
    someOtherNode.eval(someAttribute);
    ....}
    values.set(attr,resultValue);
    working.set(attr,false);
    return resultValue;
}
```

The algorithm terminates if all attributes which are reachable from the root could have been evaluated - unreachable attributes are ignored. It also detects cycles in the moment when the method body is entered a second time before having the corresponding attribute evaluated.

L-attributed grammars
L-attributed grammars belong to a class of AGs which can be evaluated during syntax analysis. They do not require an explicit construction of the syntax tree. Instead attribute values are annotated to elements on the parser stack. Hence compilers which use this technique typically become very efficient. Unfortunately some restrictions on how attributes are allowed to be defined must be introduced. The well-established deterministic parse algorithms for LL and LR grammars 'traverse' on the parse-tree from top-down in a depth-first manner. Hence attributes have to be evaluated in the same order. This means that a node n in the parse-tree (or grammar-symbol which is currently derivated) should only carry attributes which directly depend on synthesised attributes of nodes which are left siblings of n. Of course it may depend on inherited attributes of its father and on the synthesised attributes of its direct children too. Figure 6 shows such a situation for a production $p = (X_0 \rightarrow X_1 \dots X_i \dots X_{np})$.

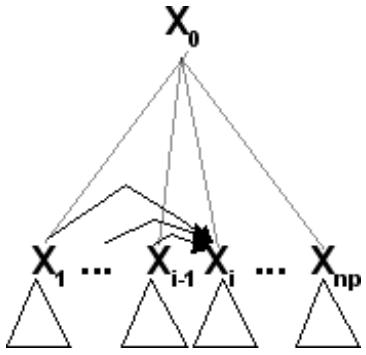

Figure 6: allowable attibute flows from siblings to X_i

More formally a normalized attribute grammar G is called $L{-}attributed$ if for each $p \in P$ with $p = (X_0 \rightarrow X_1 \dots X_{np})X$ and for every $a \in INH_{X_j}(1 \leq j \leq np)$ the semantic rule $f_{(p,j,a)}$ has arguments $b \in (\bigcup SYN_{X_k})(1 \leq k \leq j-1)$ or $b \in INH_{X_0}$ only.

LL-attributed grammars
L-attributed grammars having an underlying context-free LL(1) grammar are called LL-attributed grammars.
Hence it is possible to extend the standard parsing techniques for LL(1). In the table driven approach a parser holds only minimal information per step. This information has to be enriched by a kind of label which marks when all symbols in a production have been completely derived and the synthesised attributes of the production's root X_0 can be evaluated. This might be done by a special symbol placed behind a production on the parser stack. Additionally the attributes themselves have to be stored somewhere because after every shift step a grammar-symbol is removed or replaced from the parser stack. Hence the parse tree has to be built-up somehow in parallel, at least temporary. For instance symbols on the stack could be associated with production elements in the derivation queue which such a parser normally builds up. There attribute values could be stored and later be read. An other option would be an explicit partial build-up of the parse tree with nodes currently needed only.
A recursive descent parser for LL(1) would have all these things included automatically. Such a parser can easily be constructed from recursive functions which directly correspond to the grammar productions. Due to the recursive calls a partial parse tree is implicitly constructed on the runtime-stack. A production has been completely derived when a function returns.
Unfortunately our AG AG_{hex} is not a LL(1) grammar. So we use only the right recursive part of our grammar to demonstrate how to create an attribute evaluator:

attribute grammar AG_{hexLL1}
...
rules

r1: $NUMBER \rightarrow NUM$
$NUMBER.dvalue = NUM.dvalue$
$NUM.position = 1$

r2: $NUM \rightarrow HEX\ NUM$
$NUM_0.dvalue{=}HEX.dvalue{*}16^{-NUM_0.position} + NUM_1.dvalue$
$NUM_1.position = NUM_0.position + 1$

r3: $NUM \rightarrow \epsilon$
$NUM.dvalue = 0$

r4: $HEX \rightarrow \{0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F\}$
$HEX.dvalue = getValueForChar()$

With this we can construct a very simple recursive descent LL(1) parser.

```
public real number(){
```

```
    int position=1;
    return num(position);
}
public real num(int position){
    if(getLookAhead().isHexValue()){
        return hex()*pow(16,-position)+num(position+1);
    }
    else if(getLookAhead().isEndOfStream()){
        return 0;
    }
    throw new Exception("Unexpected character");
}
public real hex(){
    return getValueForChar();
}
```

As one can see it seems to be very intuitive to create such a parser. It might be a suitable solution for small problems. Creating a L-attributed LR(1) is much more complicated, because such parsers do not explicitly 'expand' grammar productions as LL(1) parsers do. Additionally there might be unresolvable conflicts between different items in a parser state on how to compute inherited attributes of another item. So we do not discuss LR attributed grammars here. It just should be said that a SLR(1),LR(1) or LALR(1) grammar possibly might not be L-attributable!

S-attributed grammars
S-attributed grammars are a subclass of L-attributed grammars containing synthesised attributes only. Hence there are no more restrictions concerning inherited attributes. So a S-attributed LL(1) grammar can always be called LL-attributed as well as a S-attributed LR(1) can always be called LR-attributed.

Ordered attribute grammars
This type of attribute grammars can be evaluated using a statically precomputed total order for the attributes of every grammar symbol of an AG. An evaluator visits every node in an abstract syntax tree multiple times. During a visit, at least one attribute of a node has to be evaluated. A *visit oriented evaluator* enters a node in the tree after it has computed one or more dependent values in the upper tree. Then it computes some values of attributes in that node which might be important for lower nodes and descents to visit some of the lower nodes. This is called a sequence of visits. Such a sequence can be determined by regarding the attribute dependency graphs of grammar symbols. The set $s(X)$ from the above cycle test can be used. Unfortunately $s(X)$ only contains attribute flow from inherited attributes of a grammar symbol X to synthesised attributes of X. So it only has information about the attribute flow through subtrees in which X is label of the root. For computing a complete attribute dependency information $R(X)$ one has to consider the upper attribute flow $t(x)$ too. The computation of $t(X)$ is described in [2] and will not be regarded here. $R(X)$ is defined as $R(x) = s(X) \cup t(X)$. Figure 7 shows $R(NT)$ for a hypothetical non-terminal NT with attributes $INH_{NT} = \{x\}$ and $SYN_{NT} = \{y, z\}$.

Each i-th visit Vi_X of nodes N with $X = label(N)$ consists of a set of inherited attributes and a set of depending synthesised attributes. These attributes will be evaluated during a visit by computing the inherited ones first. For instance

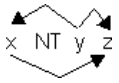

Figure 7: $R(X)$ for a non-terminal NT

$V1_{NT}$ of our short example would contain $V1_{NT}.INH = \emptyset$ because y in $V1_{NT}.SYN = \{y\}$ does not depend on any inherited attribute of NT. After the second visit $V2$ with $V2_{NT}.INH = \{x\}$ and $V2_{NT}.SYN = \{z\}$ the attributes of NT are completely evaluated.
Typically visit oriented evaluators are implemented as a recursive procedure on the nodes of the AST. Note that it might not be possible to compute a sequence of visits for a grammar out of the $R(X)$. Unfortunately this can not easily be checked,e.g. it can not be checked by introducing some syntactical restrictions as L-attributed grammars do.

5. CONCLUSION

Attribute grammars seem to be a good choice for defining the static semantics of a context free language. The concept of inherited attributes and synthesised attributes is quite simple and easily applicable.
Attribute evaluators can be generated automatically, but not all methods can be used for that. For instance ordered attribute grammars and LR-attributed grammars introduce restrictions which are not only of syntactical kind. Hence those methods might be difficult to use for developers who do not have in-depth knowledge in compiler construction. So in most cases and for small maybe domain oriented languages it would be better to use more simpler methods like the demand-driven approach or LL-attributed grammars.

6. REFERENCES

[1] Donald E. Knuth: *Semantics of Context-Free Languages,* Mathematical Systems Theory Vol. 2, No. 2 (1968)

[2] R. Wilhelm, D. Maurer: *Uebersetzerbau,* ISBN 3-540-61692-6 (Springer, 1997)

[3] Lothar Schmitz: *Syntaxbasierte Programmierwerkzeuge,* out of print (Teubner, 1995)

[4] Donald E. Knuth: *The genesis of attribute grammars,* Proceedings of the international conference on Attribute grammars and their applications, 112 (1990)